10
Convolutional Nets

This chapter is a brief introduction to convolutional deep nets, which revolutionized computer vision as well as audio processing.

The computational elements between any two layers are arranged in a special way, with the *same* weight appearing in several edges across the layer. This architecture is described below.

10.1 The Convolutional Primitive: Shared Weights and Equivariance

The fundamental flaw of applying a standard fully-connected network (MLP) to image data is its disregard for spatial structure. An MLP treats an input image as a high-dimensional vector, and as such, it must learn from scratch that pixels nearby are more related than pixels far apart. Furthermore, if an object of interest appears in a different location, the network has no innate mechanism to recognize it as the same object; it must learn this invariance from vast amounts of data. Convolutional Neural Networks (CNNs) are designed to overcome these limitations by building in two powerful priors: **locality** and **translation equivariance**.

To formalize this, we must first move from the vector-based notation of MLPs to the tensor-based notation of CNNs. An input image is not a vector in $\mathbb{R}^d$, but a 3D tensor $X \in \mathbb{R}^{H \times W \times C_{in}}$, where H is the height, W is the width, and C_{in} is the number of input channels (e.g., $C_{in} = 3$ for an RGB image). We denote a specific element of this tensor by $X_{i,j,c}$, where $i \in \{1, \dots, H\}$, $j \in \{1, \dots, W\}$, and $c \in \{1, \dots, C_{in}\}$.

10.1.1 The Convolution Operation

The core of a CNN is the convolution layer, which replaces the matrix multiplication of a fully-connected layer with the convolution operation. This operation uses a small, learnable filter called a **kernel** to

extract local features.

Let's first consider the simplest case: a grayscale image ($C_{in} = 1$) and a single kernel that produces a single output feature map ($C_{out} = 1$). The kernel K is a small matrix of learnable weights, $K \in \mathbb{R}^{K_H \times K_W}$. The convolution operation slides this kernel over the input image and, at each location, computes the element-wise product between the kernel and the overlapping input patch, summing the result. The output is a 2D **feature map** Z. The value at position (i, j) in the output map is given by:

$$Z_{i,j} = (X * K)_{i,j} = \sum_{k=1}^{K_H} \sum_{l=1}^{K_W} X_{i+k-1,j+l-1} \cdot K_{k,l} \tag{10.1}$$

This operation is performed for all valid positions (i, j), producing a feature map that highlights where the pattern encoded by the kernel appears in the image.

10.1.2 An Intuitive Example: Hand-Crafted Kernels

To make this concrete, consider a simple 3×3 kernel designed to detect vertical edges in a grayscale image.

$$K_{vert} = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} \tag{10.2}$$

When this kernel is convolved with the image, its structure acts as a pattern detector. The positive values on the left and negative values on the right compute a localized horizontal difference. If the kernel is over a region of flat color, the positive and negative terms will cancel, and the output $Z_{i,j}$ will be close to zero. However, if it is centered on a vertical edge (e.g., a bright region to the left of a dark region), the output will be a large positive number. Similarly, a horizontal edge detector can be constructed as:

$$K_{horiz} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix} \tag{10.3}$$

This idea extends naturally to multi-channel inputs like RGB images. Suppose we want to build a filter that responds specifically to *vertical red edges*. Our input has three channels (R, G, B), so our filter must also have a component for each channel. For a single output, the kernel K would be a $3 \times 3 \times 3$ tensor. We can design it as follows:

- **Red Channel Kernel:** A vertical edge detector to find gradients in red intensity.

- **Green & Blue Channel Kernels:** Zero matrices, to ignore information from these channels.

$$K_R = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}, \quad K_G = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad K_B = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{10.4}$$

The total activation is the sum of the convolution results from all three channels. This filter will only produce a strong positive output in regions where there is a sharp vertical gradient in the red channel specifically.

The intent behind CNN architecture is to take away the need to hand-craft such kernels. Instead, the values of the kernel tensor K are learnable parameters of the model, and the network learns the most useful feature detectors for the task at hand through backpropagation.

10.1.3 Generalizing the Convolution Layer

In the general case, we have C_{in} input channels and want to compute C_{out} output channels. The layer's weights are thus represented by a 4D tensor $K \in \mathbb{R}^{K_H \times K_W \times C_{in} \times C_{out}}$. [1] The pre-activation output $Z \in \mathbb{R}^{H_{out} \times W_{out} \times C_{out}}$ is computed as follows. The value for the m-th output channel (feature map) is obtained by summing the results of convolving each input channel with its corresponding kernel:

[1] You can think of this as having C_{out} different filters. Each of these filters is composed of C_{in} kernels, one for each input channel. The m-th filter, $K_{:,:,:,m}$, produces the m-th output map.

$$Z_{i,j,m} = \left(\sum_{c=1}^{C_{in}} X_{:,:,c} * K_{:,:,c,m} \right)_{i,j} = \sum_{c=1}^{C_{in}} \sum_{k=1}^{K_H} \sum_{l=1}^{K_W} X_{i+k-1,j+l-1,c} \cdot K_{k,l,c,m} \tag{10.5}$$

A learnable bias term b_m is added to each element of the m-th output map, and a non-linear activation function $\sigma(\cdot)$ (such as ReLU) is applied to produce the final output tensor Y:

$$Y_{i,j,m} = \sigma(Z_{i,j,m} + b_m) \tag{10.6}$$

Stride and Padding. Two key hyperparameters control the behavior of the convolution and the dimensions of the output tensor.

- **Padding (P):** It is common to add a border of zeros around the input tensor. This allows the kernel to be centered on pixels at the edge of the image and can be used to preserve the spatial dimensions of the input. With P pixels of padding, the effective input dimensions become $(H + 2P) \times (W + 2P)$.
- **Stride (S):** The stride dictates the step size the kernel takes as it moves across the input. A stride of $S = 1$ means moving one pixel at a time; $S = 2$ means skipping every other pixel.

Incorporating these parameters, the relationship between input dimensions (H_{in}, W_{in}) and output dimensions (H_{out}, W_{out}) for a kernel of size $K_H \times K_W$ is:

$$W_{out} = \left\lfloor \frac{W_{in} - K_W + 2P}{S} \right\rfloor + 1, \quad H_{out} = \left\lfloor \frac{H_{in} - K_H + 2P}{S} \right\rfloor + 1 \quad (10.7)$$

And the explicit formula for the convolution becomes:

$$Z_{i,j,m} = \sum_{c=1}^{C_{in}} \sum_{k=1}^{K_H} \sum_{l=1}^{K_W} X'_{S(i-1)+k,S(j-1)+l,c} \cdot K_{k,l,c,m} \quad (10.8)$$

where X' denotes the padded input tensor.

10.1.4 Pooling layers

The convolution operation produces feature maps that are equivariant to translation; if an input object shifts, the corresponding activation in the feature map also shifts. To build a representation that is robust to small variations in the object's position and to progressively reduce the spatial dimensionality of the network, convolutional layers are often followed by a **pooling layer**. A pooling layer operates on each feature map independently, replacing a local patch of the map with a single summary statistic.

The most common type is **max-pooling**. It typically takes a 2×2 window and, with a stride of $S = 2$, slides it across the feature map, outputting only the maximum value from each window. Formally, for an input feature map Z and a $K \times K$ pooling window with stride S, the output Y is given by:

$$Y_{i,j} = \max_{0 \le k,l < K} Z_{S(i-1)+1+k,S(j-1)+1+l} \quad (10.9)$$

The intuition is powerful: the exact location of a feature within a small local region becomes less important than its maximal strength. If a vertical edge detector fires strongly anywhere within a 2×2 patch, the max-pooling layer reports that strong signal, effectively creating a small degree of local translation invariance. An alternative, **average-pooling**, computes the mean of the values in the window, providing a smoother summary. By interspersing convolutional layers with pooling layers, a typical CNN architecture progressively shrinks the spatial dimensions (H, W) while often increasing the number of feature channels (C), creating a hierarchy of increasingly abstract and spatially robust representations.

10.1.5 Backpropagation on convolutional nets

The backpropagation algorithm can be used to write the gradient of the training loss. The fact that the same parameter may be reused

multiple times within the same layer —so-called "weight tying"– requires a small modification so that gradient values from all copies of a shared parameter are pooled to get the update. See Chapter 3.

10.2 *A Deeper Look: Why are convolutional nets sample-efficient?*

We have seen above that the convolutional architecture seems a natural fit for image data. In practice (especially for vision and audio-processing tasks) they are able to learn with much less data.

Usually this sample-efficiency is understood by pointing to the fact that the convolutional structure captures translation-invariance, and hence has the right "inductive bias" for this setting. This intuition, however, faces a simple paradox: a sufficiently wide fully-connected (FC) network can perfectly simulate any ConvNet. When the FC net is being trained, the training is in principle free to zero out some weights and also achieve weight-sharing by assigning the same values to groups of weights. So it is in principle able to express/simulate a convolutional net. Why, then, do FC nets struggle with image tasks in practice, requiring vastly more data for similar performance?

Now we provide a more rigorous answer[2]. The key insight is that the sample efficiency gap arises not from the architecture's expressiveness alone, but from the subtle interplay between the architecture and the symmetries respected by the training algorithm.

[2] Why are Convolutional Nets more Sample-Efficient than Fully-Connected Nets? Li, Zhang, and Arora (ICML 2021)

The Symmetries of Standard Learning Algorithms. Most standard training algorithms for FC nets, including SGD with random Gaussian weight initialization, possess a hidden symmetry: **orthogonal equivariance**. An algorithm A is orthogonally equivariant if, for any orthogonal matrix R, training on a rotated dataset yields a rotated version of the function learned on the original dataset. The algorithm has no inherent preference for any coordinate basis.

Problem 10.2.1. *Prove the above assertion. (Hint: If a fixed rotation R is applied to every input x then the loss $\ell(x)$ becomes $\ell(Rx)$. How does the new gradient behave?)*

A Task to Separate ConvNets from FC Nets. Consider the task of learning the function $h^*(x) = \text{sign}(\sum_{i=1}^{d/2} x_i^2 - \sum_{i=d/2+1}^{d} x_i^2)$. where $P_x = \mathcal{N}(0, I)$ is the data distribution. A simple ConvNet can learn this to high accuracy (say, 95%) with $O(1)$ samples.

Problem 10.2.2. *Describe such an architecture. (Note that it can have the hardwired prior that all coordinates are to be squared.)*

Now show that for an FC net trained with an orthogonally equivariant algorithm, this task requires $\Omega(d^2)$ samples. We use a geometric packing argument.

The Epsilon-Packing Argument. The proof strategy is to construct an exponentially large set of functions that are all far from each other, and then show that a successful learning algorithm must use many samples to distinguish between them.

1. **Define a function class and distance.** The function h^* belongs to the class of functions generated by its rotations: $\mathcal{H}_{rot} = \{h^* \circ R \mid R \in O(d)\}$, where $O(d)$ is the group of $d \times d$ orthogonal matrices. We measure the distance between two functions on the data distribution $P_x = \mathcal{N}(0, I)$ as their probability of disagreement:
$$dist(f_1, f_2) = \mathbb{P}_{x \sim P_x}[f_1(x) \neq f_2(x)]. \tag{10.10}$$
2. **Construct an ϵ-packing.** Our goal is to find a very large set of functions $\{f_1, f_2, \ldots, f_M\} \subset \mathcal{H}_{rot}$ such that for any pair f_i, f_j with $i \neq j$, $dist(f_i, f_j) \geq \epsilon$. Such a set is called an ϵ-packing.

From Function Distance to Matrix Distance. The first key step is to relate the distance between functions to the geometric distance between the rotation matrices that define them. The function h^* can be written as $h^*(x) = \text{sign}(x^T Q x)$, where Q is a diagonal matrix with $+1$ for the first $d/2$ entries and -1 for the others. A rotated function is then $(h^* \circ R)(x) = \text{sign}(x^T (R^T Q R) x)$. The distance between two such functions is governed by the difference between the matrices $R_1^T Q R_1$ and $R_2^T Q R_2$. A non-trivial result from high-dimensional probability shows that for some constant c, this simplifies to:
$$dist(h^* \circ R_1, h^* \circ R_2) \geq c \cdot ||R_1 - R_2||_F \tag{10.11}$$
where $||\cdot||_F$ is the Frobenius norm. This result is crucial: it means a set of geometrically separated rotation matrices corresponds to a set of statistically separated functions.

Problem 10.2.3. *The relation in (10.11) is the technical core of the proof. Let's build intuition for it. Two functions $h^* \circ R_1$ and $h^* \circ R_2$ disagree on input x if the signs of the quadratic forms $x^T(R_1^T Q R_1)x$ and $x^T(R_2^T Q R_2)x$ are different.*

(i) Show that this condition is equivalent to the sign of their product being negative: $(x^T(R_1^T Q R_1)x) \cdot (x^T(R_2^T Q R_2)x) < 0$.

(ii) While a full proof is complex, explain intuitively why the probability of this event for $x \sim \mathcal{N}(0, I)$ should increase as the matrices R_1 and R_2 become "more different" from each other. [3]

[3] Hint: Consider the angle between the vectors $v_1 = (R_1^T Q R_1)x$ and $v_2 = (R_2^T Q R_2)x$. The probability of disagreement relates to the probability that the dot product $v_1 \cdot v_2$ is negative.

The High-Dimensional Geometry of Rotations. With the relation from (10.11), our task reduces to a purely geometric one: find a large packing of rotation matrices. The key insight is that the manifold of orthogonal matrices $O(d)$ is a high-dimensional space. To specify an arbitrary rotation requires specifying $D = \frac{d(d-1)}{2} = \Omega(d^2)$ parameters.

We can now use a volume argument. Let's find the maximum number of matrices M such that $||R_i - R_j||_F \geq \delta$ for all $i \neq j$. Each matrix in this packing can be thought of as the center of a small "exclusion ball" of radius $\delta/2$. All these balls must fit within the total space without overlapping. The number of balls M we can pack is roughly the ratio of the total volume to the volume of a single ball:

$$M \approx \frac{\text{Volume(Total Space)}}{\text{Volume(Exclusion Ball)}} \propto \left(\frac{C}{\delta}\right)^D \tag{10.12}$$

Substituting the dimensionality $D = \Omega(d^2)$, we find that we can construct an exponentially large packing:

$$M \geq \left(\frac{C}{\delta}\right)^{\Omega(d^2)} = \exp(\Omega(d^2)) \tag{10.13}$$

This means there are exponentially many rotation matrices that are all geometrically far from each other.

The Information-Theoretic Conclusion. The orthogonally equivariant learning algorithm faces an enormous challenge.

1. There exists a set of $M = \exp(\Omega(d^2))$ candidate functions, all of which are mutually ϵ-distinct on the data distribution.
2. The learning algorithm is given one of these functions as the ground truth and must identify it using only labeled samples (x_i, y_i).
3. Since the label y_i is binary, each sample provides at most one bit of information. To distinguish the true function from the $M - 1$ other possibilities, the algorithm needs to acquire at least $\log_2(M)$ bits of information.

Therefore, the number of samples n must satisfy the information-theoretic lower bound:

$$n \geq \log_2(M) = \log_2(\exp(\Omega(d^2))) = \Omega(d^2) \cdot \log_2(e) = \Omega(d^2) \tag{10.14}$$

The FC net's orthogonal equivariance forces it to solve a problem in an $\Omega(d^2)$-dimensional space. The sheer size of this space, containing an exponential number of distinct hypotheses, is what requires an $\Omega(d^2)$ sample complexity to resolve the ambiguity. The ConvNet, by breaking this symmetry, avoids this curse.